Compactação e descompactação de dados utilizando a árvore digital de Huffman: Implementação

Thiago Galves Moretto[1]

[1]Universidade Católica Dom Bosco
Centro de Ciências Exatas e da Terra
Curso de Engenharia de Computação
Campo Grande, Brasil, 79117-900

thiago@moretto.eng.br

Abstract. *Describes the implementation of a file compressor and decompressor aplication using the technique of the digital tree of Huffman, got resulted safisfactory however the technique does not have a good one being of compacting.*

Resumo. *Descreve a implementação de um aplicativo compactador e descompactador de arquivos utilizando a técnica da árvore digital de Huffman, obteve resultados satisfatórios porém a técnica não tem um bom poder de compactação.*

1. Introdução

Compactar dados é um ramo extramente importante, informações compactas demoram menos tempo em tráfego, seja pela Internet, ou uma transmissão feita por ondas de rádios, transmissões de satélite, enfim, a necessidade para diminuir a informação e poder recuperá-la é o que faz com que a compactação e por consequencia a descompactação é um dos ramos da computação muito estudados. Em 1954, o graduando do MIT David Huffman[1] elaborou uma técnica bastante simples para compactação e é nesta técnica que se basea este trabalho, na implementação para criar um compactador e descompactador com bastante eficiência e velocidade, para isto também existem as estruturas de dados que auxiliam na criação destes códigos para executar determinada tarefa com tempos de buscas, inserções e remoções com velocidade.

1.1. Objetivos

Implementar a técnica da criação da árvore binária de huffman para viabilizar a compactação e posterior descompactação de arquivos com eficiência e velocidade, ela deve ter no mínimo as seguintes caracteristicas:

- Ser capaz de compactar qualquer tipo de arquivo.
- Poder descompactar os arquivos de forma com que fiquem que nem os originais.
- Somente utilizar o método a árvore binária de Huffman.

1.2. Metodologia

Utilizar estruturas de dados com tempo de buscas mínimo possível para que o aplicativo esteja o máximo otimizado e tenha eficiência em arquivos e alfabetos grandes, essa minimização de tempo é auxiliada com algumas características da liguagem Java que

[1]DAVID A. HUFFMAN, Professor formado em Engenharia Elétrica pelo MIT (Massachusetts Institute of Technology)

possibilitam uma vasta gama de classes de entrada e saídas de fluxo ou I/O e o suporte a *buffer*.

A liguagem Java traz portabilidade o que viabiliza uma melhor utilização da aplicação e diferentes plataformas.

2. Fundamentação teórica

2.1. Algoritmo de Huffman

A idéia do algoritmo é que os simbolos tenha tamanhos variados e não fixos, a técnica cria uma tabela em que cada simbolo tenha seu conjunto de bits, assim os que tenham mais ocorrências na entrada tenha um código menor possível, a determinação deste código e feita pelo algoritmo discutido a seguir.

Para que o algoritmo elabore este conjunto de bits para cada simbolo é necessária fazer uma análise da entrada e obter quais são os simbolos e suas ocorrências, é necessário termos o conceito de árvore digital binária[2].

Com as ocorrências em mãos é necessário citar a instituição do nó da árvore, nela contém o símbolo ou seu código *ASCII*[3] e sua ocorrência. Com simbolos encontrados na entrada e sua ocorrência é criado uma raiz da árvore digital para cada com as devidas informações. Em nosso caso visando desempenho os nós foram armazenados em uma *Heap*[4] (ou lista de prioridade) onde a ordenação é do menor para o maior e para cada remoção, o menor do Heap é removido e retornado.

Com o *Heap* conténdo as raiz o seguinte algoritmo é capaz de constituir a árvore digital constituindo os conjuntos de bit de cada simbolo, lembrando que as informacões só ficam nas folhas da árvore.

```
enquanto número de elementos da heap for maior que 1 faça
  retire o próximo elemento da heap e armazene em T1
  retire o próximo elemento da heap e armazene em T2
  some as ocorrências de T1 e T2 e coloque em T3
  faça de T1 filho esquerdo de T3
  faça de T2 filho direito de T3
  adicione T3 na heap
fim enquanto

retire o próximo elemento da heap e coloque em T
```

Após a execução a árvore digital final estará em *T*, o seguinte algoritmo recursivo percorre a árvore e insere os códigos presentes em uma tabela Hash[5] em que a função de dispersão é o próprio código *ASCII* do simbolo.

```
função constroi_table_de_huffman ( nó, codigo_corrente )

    se nó esquerdo é nulo e nó direito é nulo então
      tabela_de_huffman [ código ASCII ] <- codigo_corrente
      retorna
    fim se
```

[2]http://www.nist.gov/dads/HTML/digitaltree.html
[3]http://pt.wikipedia.org/wiki/ASCII
[4]http://en.wikipedia.org/wiki/Heap
[5]http://en.wikipedia.org/wiki/Hashtable

```
    se nó esquerdo não for nulo
      constroi_table_de_huffman ( nó esquerdo, codigo_corrente + '0' )
    fim se

    se nó direito não for nulo
      constroi_table_de_huffman ( nó direito, codigo_corrente + '1' )
    fim se

  fim constroi_table_de_huffman
```

Estes dois algoritmos são os principais do aplicativo, a próxima etapa é a leitura do arquivo e símbolo por símbolo fazer a substituição por seu conjunto de bits, este etapa utiliza entradas de fluxos ou *Stream*, pode ser implementada de qualquer forma, não envolve nenhuma outra técnica específica, fica ao critério do desenvolvedor e da necessidade.

3. Desenvolvimento

3.1. Implementação

As etapas de desenvolvimento seguiram as técnicas de *Huffman*, na primeira versão não foi utilizada da estrutura de dados *Heap* e inclusão desta fez com que o tempo de construção da árvore caísse de alguns segundos para alguns milisegundos. Estes dados podem ser vistos em mais detalhes em Análise de Resultados.
O compactador constitui os seguintes elementos:

- Analise de caracteres e o número de suas ocorrências
- Criação do cabeçalho de informações
- Criação dos nós e adição na *Heap*
- Algoritmo da criação da única árvore digital
- Algoritmo recursivo para criação da tabela *Hash* dos códigos
- Leitura do fluxo de entrada e troca dos simbolos por seus códigos

O descompactador segue o processo de reversão do que o compactador fez e cria a árvore digital utilizando os mesmo métodos do compactador, o cabeçalho de informações contém o símbolo e suas ocorrências e assim sendo o processo é todo semelhante até o ponto da leitura do fluxo e descida na árvore para substituição do símbolo.

O algoritmo abaixo na linguagem *Java* é a leitura do fluxo de entrada e a substituição o conjunto de *bits* lido e a descida na árvore para a substituição pelo símbolo correto, esse é o processo fundamental da descompactação. Declarações de variáveis e alguns comentários foram ignorados.

```
while ((count = in.read(data)) != -1)
{
   for (int i=0; i < count; i++)
   {
        r = (data [i] < 0) ? data[i] + 256 : data[i];

        for (int k=0; k <= 7; k++)
        {
                if (offset == 0) // offset
                {
                   if (it.Right == null && it.Left == null) {
```

```
                        out.write((byte) it.symbol);
                        it = hf.root; // volta para a raiz
                    }

                    s = (r & (base >> k)) == (base >> k) ? 1 : 0;

                    if (s == 1) {
                        it = it.Right;
                    }
                    else {
                        it = it.Left;
                    }
                } else  {
                    offset --;
                }
            }
        }
    }
```

3.2. Arquitetura

A arquitetura do aplicativo não é complexa, há apenas uma interface com o usuário e esta é discutida mais à frente e no código não fica nada relacionado com o bruto da aplicação, tudo para que a parte de fundamental, compactação e descompactação seja aclopável com qualquer outro tipo de interface, modo texto, *Web*, e qualquer outro tipo. Tanto é isso que quem passa os fluxos de entrada e saída é o código mais próximo com interface com o usuário, por que isso? o algortimo não precisa saber de onde vem estes fluxos, apenas se pode manuzeálos, desta forma uma entrada pode ser um texto digitado em um formulário *Web*, pode estar em uma entrada de uma caixa de texto de um applet ou de um stream do tipo *Socket*, mas essa liberdade traz um problema, o algoritmo precisa saber o alfabeto completo e suas ocorrências completas, isso deve ser determinado, a entrada é dada e depois o fluxo deve ser fechado e só assim o algoritmo pode ter a resposta em mãos e ela é dada no fluxo de saída. Em nosso aplicativo a manipulação é feita em fluxos de arquivos.

A parte fundamental do aplicativo fica principalmente em três arquivos, *Huffman.java*, *HuffmanEncode.java* e *HuffmanDecode.java*, com foco em *Huffman.java*, ondem ficam os algoritmos de *Huffman* e outros métodos fundamentais, *HuffmanEncode.java* há o método *compress()* responsável pela chamada de métodos quem estão na classe *Huffman* (do arquivo *Huffman.java*) para a criação da árvore digital e a tabela de códigos dos simbolos, assim, classe *HuffmanEncode* (do arquivo *HuffmanEncode.java*) fica na responsabilidade de criar e gravar o cabeçalho do arquivo e tratar os fluxos de entrada, fazer a substituição dos caracteres pelos seus respectivos códigos e enviar os *bytes* ao fluxo de saída.

Na classe *HuffmanDecode* (do arquivo *HuffmanDecode.java*) é semelhante, há o método *decompress()*, responsável pela análise do cabeçalho e o tratamento dos fluxos, onde a entrada é o arquivo compactado e a cada *bit* lido é feita uma descida na árvore até a chegada de uma folha, onde é assim substituido pelo símbolo da folha e um ”ponteiro” retorna para a raiz.

3.3. Interface com o usuário

Deve ser simples, básica e intuitiva, o usuário só quer compactar ou descompactar, há um menu em cascata que dá algumas configurações de buffer dos fluxos e prioridade de

processamento além de *pop-up's* de informações de autor e de como manuzear, compactar um arquivo é simples, através do componente *JFileChooser* o usuário escolhe o arquivo entrada que é de onde o aplicativo abrirá o fluxo de entrada e depois outra tela de seleção de arquivo ele selecionará o arquivo de saída, já existente ou não, para que o aplicativo abra o arquivo e escreva no fluxo de saída ou até mesmo crie o arquivo para após iniciar a gravação do resultado.

Há uma versão por linha de comando com duas opções, compactar e descompactar, configurações de buffer e prioridade não são configuráveis nesta versão.

3.4. Elementos Java

A parte gráfica que estabelece a interface com o usuário é feita com biblioteca *Swing* do pacote *javax.swing*, não foi utilizada nenhuma biblioteca que não pertença à *J2SE*[6], classes de entrada e saída foram do pacote *java.io*.

Os classes fluxos de entrada e saida utilizadas foram a *InputStream*, *OutputStream* e também a *DataInputStream* para facilitar na formatação das entradas.

4. Análise de resultados

Em testes com arquivos textos, imagens, binários, o aplicativo teve 100% de eficiência, conseguiu compactar todos e descompactar todos, a informação relevante é o tempo que ele leva pra realizar o trabalho, dependendo da prioridade dada ao processo o tempo de descompactação teve leve mudanças, na configuração do buffer também houve pequenas variações, segue duas tabela demonstrando a partir de vários arquivos de váriados tamanhos e alfabetos o resultado do processamento do tanto de compactação quanto compactação.

O tempo é em *milisegundos* e o *buffer* é expresso em bytes.

$Teste$	$Bytes$	$Prioridade$	$Buffer$	$Compac.$	$Descompac.$
1	29.391	Mínima	64	47	125
2	10.380.480	Mínima	64	8031	26984
3	1.926.048	Mínima	64	1219	5282
4	1.440.054	Mínima	64	719	4094

Aumentamos o buffer para 512 bytes e prioridade para máxima para os mesmo testes.

$Teste$	$Bytes$	$Prioridade$	$Buffer$	$Compac.$	$Descompac.$
1	29.391	Máxima	512	47	110
2	10.380.480	Máxima	512	5328	23735
3	1.926.048	Máxima	512	985	4140
4	1.440.054	Máxima	512	672	3016

Não houve expressivas mudanças de tempo, mas devem ser consideradas, percebe-se que em arquivos maiores essa diferença no tempo tende a se tornar sim expressiva.

5. Conclusão

Os resultados, não asseguro que eles sempre serão tão ótimos pra todos os casos, nem se serão o que deveriam ser, mas foram ótimos, no propósito de criar uma aplicação que

[6]http://java.sun.com/j2se

implemente um algoritmo e que este faça com uma certa eficiência e com uma boa velocidade os resultados são aparentemente uma amostra de que não é tão obscuro fazer uma compactação e seu processo reverso, mas atualmente já há muita aplicação, técnicas e algoritmos muito mais eficientes, à caráter de estudo e na discussão de que as estruturas de dados estão para a computação quanto o motor esta para o veículo, elas estão para melhorar a eficiencia e realmente fazer com que se obtenha o máximo de processamento, memória de uma máquina, uma computação sem estruturas de dados seria um carro puxado por dois burros, se quisessem mais potência, que se compre mais um burro. O futuro de um trabalho deste já esta no nosso presente, eficientes algoritmos nos permitem fazer ótimas compactações, podem ser melhoradas, com certeza, o grande problema é que algumas são proprietários, até vamos citar o *Zip*, que com certeza é o mais utilizado, porém há outros até mais eficientes ou não mas são ótimos, *GZip*, *Rar*, *Bzip* são alguns deles e são de código aberto, podem ser estudados e melhorados livremente.

6. References

NIST - National Institute of Standards and Technology
Dicionário de Algoritmos e Estruturas de Dados
http://www.nist.gov/dads/